AFRL-IF-RS-TR-2004-351
Final Technical Report
December 2004

INTEGRITY THROUGH MEDIATED INTERFACES

University of Southern California, at Marina del Ray

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. H546

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

AFRL-IF-RS-TR-2004-351 has been reviewed and is approved for publication

APPROVED: /s/

JOHN C. FAUST
Project Engineer

FOR THE DIRECTOR: /s/

WARREN H. DEBANY, JR., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** DECEMBER 2004	**3. REPORT TYPE AND DATES COVERED** Final Jun 99 – Dec 03

4. TITLE AND SUBTITLE INTEGRITY THROUGH MEDIATED INTERFACES	**5. FUNDING NUMBERS** C - F30602-99-1-0542 PE - 62301E/63760E PR - H546 TA - 10 WU - 01
6. AUTHOR(S) Robert Balzer	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Southern California, at Marina del Ray Information Sciences Institute 4676 Admiralty Way Marina del Ray California 90292-6695	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency, 3701 North Fairfax Drive, Arlington Virginia 22203-1714 AFRL/IFGB, 525 Brooks Road, Rome New York 13441-4505	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2004-351

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: John C. Faust/IFGB/(315) 330-4544/ John.Faust@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(Maximum 200 Words)***

We created an Integrity Manager that monitors and records the tools (i.e. programs), and operations within those tools, being applied to integrity-marked data sets to provide an end-to-end audit record of all the transformations performed on the data set. This operation level audit record can be used off-line for attribution (who made a specific change and when did it occur) and on-line for authorization (who and/or which tools are allowed to make particular types of changes to an integrity-marked data set). We also use this transaction history to recreate corrupted data sets by replaying the recorded sequence of data set modifications.
We also developed a wrapper that monitors the run-time behavior of opened email attachments to ensure that these processes don't do anything harmful. It does so by detecting violations of process-specific rules establishing the acceptable (and safe) behavior of these processes relative to four resources: the file system, the system registry, inter-host communication, and process spawning. When attempted violations are detected, the user is notified, informed of the severity of the violation, and determines whether to allow or prohibit the offending operation. The violation, the user's response, and the initiating email and attachment, obtained from the email client, are logged.

14. SUBJECT TERMS Document Integrity, Document Recovery, Integrity Manager, Safe Email Attachments, Wrappers, Execution Monitor, Contained Execution, Active Content, Malicious Code	**15. NUMBER OF PAGES** 36
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

1. Scope

This contract was aimed at developing the broad set of technologies required for protecting the integrity of documents by detecting when they are corrupted and repairing them by replaying a recorded history of the user modifications to the document.

It was based on our Instrumented Connectors technology that enables external mediators to monitor and respond to program behavior and enrich or restrict its execution environment. Because these mediators are non-bypassable they can enhance the security, safety, and integrity of unmodified COTS/legacy applications.

The original proposal consisted of a single task to wrap data with integrity marks that ensure its integrity, record its processing history (user, tool, and operations), and allow it to be reconstructed from this history if it is corrupted by program bugs or malicious attack – and to demonstrate these capabilities on a major GCCS product – Microsoft Word.

During the course of this contract, it was recognized that our Instrumented Connector technology could also be used to protect a computer from malicious Email attachments and with the concurrence of our DARPA Program Manager, developing such a cyber-defense for a COTS based email client (Microsoft Outlook) and the COTS products that handle those attachments (e.g. PowerPoint, Word, Acrobat, WinZip, etc.) was added as a second task to this contract.

Each of these tasks is described in the following sections.

2. Document Integrity

2.1. Scope

Operating systems provide a variety of facilities for invoking programs and applying them to selected files. Over time, a particular data set (i.e. file, directory, or compound document) is the product of manipulations and transformations carried out by several different programs and users. However, no record of this transaction history of which users and programs have transformed the data set is collected and kept. Operating systems only limit, through access control lists, which files users can access and with what file-level rights (read-only, write, and/or execute). No limits are placed on which programs can manipulate a particular data set or how it can be transformed. Under these circumstances, no assurances can be given as to the integrity of the resulting data set.

We remedied this situation by creating an Integrity Manager that monitors and records the tools (i.e. programs), and operations within those tools, being applied to integrity-marked data sets to provide an end-to-end audit record of all the transformations performed on the data set. This operation level audit record can be used off-line for attribution (who made a specific change and when did it occur) and on-line for authorization (who and/or which tools are allowed to make particular types of changes to an integrity-marked data set). As explained below in the Restoring Corrupted Integrity-Marked Data section, we also use this transaction history to recreate corrupted data sets by replaying the recorded sequence of data set modifications.

2.2. Approach

2.2.1. Tool-Level Integrity Tracking

At the tool level, whenever a tool is being applied (invoked) on an integrity-marked data set, the Integrity Manager determines whether the user is authorized to operate that tool on this data set in its current audit-history state and validates the integrity of the data set. Once these preconditions have been satisfied, it logs the user and tool invocation in the data set’s transaction history, and allows the tool invocation to proceed. When the tool modifies the persistent copy of the data set or writes a new version of it, the Integrity Manager attaches a new cryptographic integrity mark to the resulting data set(s).

This creates a coarse end-to-end transaction history for data sets, which records which tools were invoked by which users to transform the data set. The cryptographic integrity marks applied at the completion of each tool invocation and checked when the data set is accessed by the next tool invocation ensures that the data set has not been altered outside of the tool invocations recorded in the transaction history.

In order to build this coarse transaction history the Integrity Manager responds to process-spawn, file-open, and file-close events (generated by Instrumented Connector mediators). Process-spawn events identify the invocation of tools. They also identify the invocation of “child” processes within an invoked tool. The file-open events identify the files being used by those tools, and the file-close events identify the completion of that file usage. These mediated interfaces are completely generic and are applicable to all tools.

The Integrity Manager is activated by these mediated interfaces to monitor a tool when that tool is applied to an integrity-marked data set. This can occur when the tool is first invoked (because an integrity-marked data set is one of its input parameters), or during its execution (because it accesses an integrity-marked data set). However, the Integrity Manager can also be activated to monitor tools – specially marked as “initial digitizers” – that produce data sets from external instruments that are to be integrity-marked and monitored, even though these tools never access an integrity-marked data set.

Thus, data sets are stamped with an unforgeable "integrity mark" at their initial point of digitization and all subsequent modifications to these data sets are logged, integrity marked, and tracked throughout their digital existence to provide end-to-end data integrity.

2.2.2. Operation-Level Integrity Tracking

However, this coarse transaction history provides no insight into which portions of the data set were modified by a tool or how they were transformed. To obtain that information we installed additional instrumentation on the tool and script any Application Program Interfaces (APIs) it provides to record the application level transformations it applies to the data set (to the degree this can be obtained by instrumentation and scripting without modifying the tool itself).

COTS tools vary greatly in the extent to which they expose their internal operations and state. At the most open and cooperative end of this spectrum, some tools (e.g. Visio) generate events corresponding to some or all of the application level operations they perform on a data set. These events enable programs that register interest in those events to easily track the sequence of operations being applied to the data set (at least for the subset of operations that the tool has chosen to make visible through generated events).

Many more (e.g. PowerPoint, Netscape Navigator and Microsoft Internet Explorer, document editors, document formatters) provide a scripting API that enables a program to invoke some, all, or even a superset of the functions the tool provides to users through its graphic user interface. These scripting APIs were designed to allow users and third party developers to customize and extend the capabilities of the COTS tools (e.g. by defining macros). We use them instead to track the operations being applied by the tool to the data set. In particular, we use their ability to access the current state of the data set inside the tool to detect changes in that data set.

We should also note that these scripting interfaces could be instrumented with our technology to record any commands issued by a scripting client. These scripting interfaces are ideal interfaces to instrument for operation level integrity tracking because the modifications invoked through this interface are already expressed in application level terms (and hence don't need to be translated or inferred from lower level actions) and because all modifications initiated through this interface can be captured. Such instrumentation would therefore directly provide a history of the application level modifications made through that interface. Unfortunately, at the moment, with the exception of database and other "server" applications, use of these scripting interfaces is relatively lightly used, especially compared with the dominant usage mode of interactively driving the tool through the graphic user interface. As these scripting interfaces become more prevalent we hope that vendors will begin to use them directly as the programmatic interface through which the GUI accomplishes the operations invoked by the user. When that occurs, our instrumentation technology will be able to directly record the history of application level changes invoked by the GUI on behalf of the user.

At the most closed and uncooperative end of this spectrum, most of the remaining tools utilize a graphic user interface. This interface can be generically instrumented to record user commands (such as button pushes, mouse motion, and keyboard input). This command history is complete, but is expressed at an exceedingly low level. This makes it very difficult for people or programs to analyze the history to determine whether or not a particular application level transformation is contained within this history. However, with appropriate application and tool specific knowledge, this low-level history can be mapped into an application level history comparable to that obtained from the most cooperative event-generating tools.

In particular, using GUI "spy" technology, we can extract the unique IDs for buttons, windows, and fields exchanged between the tool and the graphic user interface that enable the tool to map user actions reported by the graphic user interface into application level commands and parameters for those commands. We can then manually make the same associations in the form of mapping rules. These mapping rules are loaded into our mediators so that they can then perform the same translation of user interface actions into application level modifications as is occurring in the tool (but not being reported).

These manually encoded tool-specific mapping rules allow us to translate a mouse-click into the pushing of a particular button on the graphic display and thereby infer the invocation of the application-level command associated with that particular button. They also identify the user-supplied fields that correspond to the parameters of these commands.

Experience has shown that these unique IDs do not remain valid across new versions of the tool. Thus, when a new version is released, we must obtain the new IDs with our GUI spy and manually update the mapping rules developed for the previous version. Though manual, for the tools we've worked with to date, this release update effort has been accomplished in well under an hour.

Given this wide range of openness and level of cooperation in the set of existing and emerging COTS tools, it is clear that obtaining operation level transformations is problematic and must necessarily be done in a tool specific way. Our strategy is to provide generic instrumentation capabilities for each portion of the spectrum described above and combine them to utilize whichever combination of APIs a particular tool supports.

One particular combination is especially promising because it would support a large number of popular COTS products. This combination uses GUI instrumentation to detect the initiation of some not-yet-identified application-level modification and the scripting API to examine the resulting state produced by that operation to determine its identity. Typically, such modifications represent "direct manipulation" operations provided by the GUI (such as attaching or detaching a connector or moving an object).

In general, this requires us to maintain a model of the previous state of the data set so that we can perform a comparison to detect what has changed. However, since direct manipulation can only be invoked on objects that are visible in the GUI, only that portion of the data set must be modeled. Often, the tool greatly simplifies the search by "selecting" and/or highlighting one or more of the objects involved in the modification. If this "focus" can be detected in the scripting API, it greatly reduces the amount of state that must be compared to detect changes.

While determining which operation was invoked by examining the changes it produced could theoretically be arbitrarily difficult, we have found writing such scripting programs to be relatively easy and straightforward because the changes were small and directly related to the user's "direct manipulation" of the interface. These "operation inferring" scripting programs must however be manually constructed and are highly tool specific.

2.2.3. Data-Set Differencing: A Rejected Alternative Approach

Before leaving this section, we must note that an alternative to operation tracking exists. This alternative would determine which portions of a data set changed by performing a "Diff" on the input and output data sets – that is, it would compare the resulting data set to the original data set after the tool completed its execution.

We rejected this approach because it is nonresponsive to DARPA's goals to "track application-level modifications and automatically append an additional integrity mark reflecting the changes." Instead of tracking these changes, it would attempt to infer them from post-processing. Moreover, like Network Sniffers, it would be operating at the lowest level of representation and would need to reverse engineer differences detected at that level into higher level modifications using tool specific abstraction rules. Numerous attempts to provide such abstraction rules for textual "Diff" programs have shown how difficult such after-the-fact reverse engineering is.

Also, by viewing tools as impenetrable monolithic black boxes, this approach is regressive and reactionary and only offers restricted short-term results before the limits of this approach are reached.

In contrast, our approach exploits whatever openness a tool provides to track and record application level modifications. Even when this openness is incomplete and we must instead infer application level modifications from graphic user interface commands, we are still operating at much higher levels of abstraction than those that arise from "Diffing" the data sets.

Moreover, by showing the power of operation level monitoring and tracking, we will provide an additional source of encouragement for COTS tool vendors to further open up their products.

2.2.4. Restoring Corrupted Integrity-Marked Data

Creating a complete transaction history for integrity-marked data sets is important not only to attribute when, how, and by whom particular changes occurred, but also to provide the basis for restoring corrupted data.

Our cryptographic integrity marks (i.e. digital signatures) enable us to detect when an integrity-marked data set has been corrupted. Such corruption can occur from malicious attacks or from errant (buggy) programs that inadvertently overwrite persistent data. However such corruption occurs, the Integrity Manager will detect it when the integrity-mark is checked as a tool first accesses the data set. It should be noted that this integrity-mark check would be performed whenever any tool, including the backup program, accesses an integrity-marked data set. Thus, corrupted data sets will never be saved as backups and corruption detection will occur at least as often as backups are scheduled (presumably daily).

Rather than repairing the corrupted data set by maintaining duplicate copies (which may get very large), we automatically rebuild it from the transaction history that is already being maintained to attribute data set transformations (i.e. who made what changes when). Starting with an uncorrupted previous version of the data set (possibly the initial digitized version of the data set), one or more tool sessions are “replayed” to duplicate the data set’s recorded transaction history. A combination of simulated user inputs and API commands extracted from the transaction history drive these tool sessions.

<u>This recreation of corrupted data sets is automatic and doesn’t require users to manually determine which portions of the data set have been corrupted or how to repair that corruption. It also cuts the time of repair from the weeks now required for manual analysis and restoration to the minutes required for the tools to duplicate the data set’s modification history.</u>

2.2.5. Resisting Attacks on Integrity-Marked Data

The ability to restore corrupted data sets critically depends upon the integrity of the transaction history being maintained for these data sets. We therefore undertook a series of measures to reduce the vulnerability of these transaction histories to either malicious attacks or wayward (but non-malicious) programs.

While the integrity-marked data sets must be visible and accessible, there is no similar requirement for the transaction histories being maintained by the Integrity Manager. We can therefore hide the location of the transaction histories from both users and programs. In addition to employing the operating system’s access control facilities, we combined two separate concealment techniques to minimize the corruption vulnerability of these transaction histories.

The first concealment technique is to use our Virtual File System to hide the existence of these transaction histories. This wrapper technology encapsulates a program to intercept and mediate all of its file system operations. The mediators filter the set of files and directories returned from file/directory search, enumeration, and existence requests in accordance with a set of visibility rules. They thus create a “virtual” file system in which only specified portions of the real file system are visible – and hence – accessible.

We then hide the existence of the transaction histories by defining the Virtual File System visibility rules to filter out all files containing transaction histories, and by specifying that our Instrumentation Manager should apply this wrapper to all programs except the Integrity Manager. Thus, the Integrity Manager is the only program running on the machine that can see and access these transaction histories.

As an additional safeguard to protect these transaction histories, just in case our Virtual File System is somehow breached, we also conceal the transaction histories by employing randomization techniques to vary the name and location of these transaction histories from system to system. This will make it very difficult for malicious code to locate these transaction histories even with public access to the Integrity Manager algorithms and source code.

Finally, we also digitally sign the transaction histories so that we can detect corruption to the transaction histories despite our combined concealment techniques.

2.2.6. Target COTS Product

For this effort, we chose Microsoft Word as the main COTS product to wrap and expand into a Document Integrity Manger. We did so because of its large market share and the military’s heavy reliance on it for planning and operations (MS Word and PowerPoint are reportedly the two most widely used applications within the military). We also did so because the Word scripting interface was quite comprehensive and well documented and because it supported a reasonable set of native events. We have also experimented with PowerPoint to investigate the generality of our techniques and the effort involved in transferring some of these techniques to another application.

2.3. Accomplishments

We successfully developed an Integrity Manager for Microsoft Word that detects corrupted documents and repairs them by replaying a captured history of the user's modifications to the document.

In addition this Integrity Manager was able to identify who made particular changes to the document and when those changes were made.

While we did not achieve complete coverage of all Word commands (see Achieved Coverage subsection below), we did cover all but the most rarely used commands (according to a usage survey we conducted).

These accomplishments were realized by overcoming five technical challenges that are described in the next section.

2.3.1. Achieved Coverage

Our current implementation of the Word Integrity Manager manages to identify and replicate the invocation of 180 commands from our target set of 209 commands that modify the document textual content. Most of the remaining commands were deliberately left out because they are almost never used (according to a survey carried out in our group). Word commands can be invoked by a number of different methods, including menus, toolbar controls, keyboard shortcuts, and the object's direct manipulation (e.g., dragging paragraph margins). Our integrity manager monitor is able to recognize these commands regardless of the method used to invoke them.

2.3.2. Limitations

Although some operations are sensitive to Word's configuration (such as whether the *AutoCorrect* feature is turned on or not), we are not currently detecting and recording changes in this configuration. We are also not recording changes to the templates (including modifications to the *Styles*). Finally, although we might be able to detect the invocation of macros and invoke the same macro during document reconstruction, we wouldn't be able to reproduce the same behavior if that macro interacted with the environment (e.g., interactive input from the user or reading the contents of a file) unless those interactions were also captured.

2.4. Technical Challenges Overcome

There were five technical challenges that had to be overcome in this project. By far, the most challenging was capturing all of the changes being made to a document so that if it were corrupted the recorded history of those changes could be used to rebuild the document. The second was replaying that change history to rebuild the document. The third was detecting whether or not a document had been corrupted. The fourth was using the recorded change history to determine who made particular changes to the document and when they did so. The final technical challenge was managing the complexity of Word.

2.4.1. Capturing All Document Changes

Rebuilding corrupted documents from a recorded history of changes required the capturing of all changes to the document. This was in marked contrast to our prior work on the PowerPoint Design Editor where only four topological operations had to be captured. Our effort here was guided by the following design goals:

- **The history of changes should be described at the application level of the COTS product (MS Word) rather than at the low-level GUI interactions level**. For example, the user action of clicking the command bar *Bold* button should be described as *Toggling Bold face* and not *clicking left mouse button at position (x,y)*. While recording the low-level GUI interactions would be simple – because the Windows operating system already provides a journaling mechanism for capturing these low-level GUI interactions – doing so would make the history much larger and preclude its use for attribution.
- **Changes should be described independently from the particular mechanism in which those changes were performed**. The same action can be performed by several mechanisms. For example, the *Toggle Bold face* action might be invoked by at least three different mechanisms: clicking on a command bar button, selecting it from a menu, and through a keyboard shortcut. These alternative invocation methods are semantically equivalent and representing them as different actions would only add unnecessary complications.
- **The history of changes shouldn't include actions that don't modify the document.** Viewing actions like browsing the document or changing the zoom control don't affect the content of the document and hence should be filtered out of the change history.

The user's actions are determined by combining information from several sources including events reported by the Word and Office API, pseudo-events reported by the wrapper, comparing the content of the document before and after an action, and querying the state of the GUI.

2.4.1.1. Capture Capabilities provided by the Word API

Word's COM interface provides the following functionality that was instrumental in capturing the user's actions:

- An object model that allows the state of a document to be examined.
- Trappable events that report coarse grained operations on documents (e.g., Open, Save, Close, New, Activate window, and Deactivate window). For some of these operations the corresponding event is triggered before the operation is invoked and for others the event is triggered after the operation is completed. For example, for the *Save* and *Close* operations the corresponding event is triggered before the operation is invoked. However, for the *Open* and *New* operations the corresponding event is triggered after the operation is completed.
- Trappable events that report the activation of standard command bar controls like standard *Pushbuttons*, *Combo boxes*, and *Menu items*. For the *Pushbuttons* and the *Menu items* the corresponding event is triggered before the command associated with that button or menu item is executed. For the *Combo boxes* the corresponding event is triggered after the command associated with that combo box is executed.

- A trappable event that is triggered each time that a different region of a document is selected.

Although these capabilities were useful, they were not sufficient for capturing user actions. One reason is that some of the actions that are preceded by a trigger event, such as *Save* or *Close,* may not actually occur because the user aborts the operation when presented with the confirmation dialog box. Word's COM interface doesn't report whether such operations actually occurred or were aborted. This lack of confirmation for operations with preceding events is clearly inadequate for reliably capturing user actions.

A second limitation is that many of the user actions we needed to capture (such as typed text or manipulation of non-standard command bar controls) had neither a preceding nor a following event associated with them, and hence were invisible through the native Word API.

2.4.1.2. Augmenting the Word API with Pseudo-Events.

We therefore constructed our wrapper to generate the following pseudo-events:

- **Completion of the command associated with a pushbutton or a menu item.** This compliments the activation event provided by Word's API and allows the triggered script to examine the state of the document after the execution of the command (e.g., to determine the values entered by the user in the fields of a dialog box displayed by a command).
- **Manipulation of non-standard command bar controls** (e.g., the Undo control)
- **Typed text**
- **Keyboard shortcuts**
- **Dragging text** (e.g., to move or copy text)
- **Completion of the Save and Close operations**. This pseudo-event confirms that the user did not abort the Save or Close operation. For the Save operation it also provides access to the pathname of the saved document and allows the checksum of the saved document to be computed.

2.4.1.3. Document Modification Capture Examples

In this section we describe how a few Word operations were captured. These examples illustrate how information from different sources is combined to identify the actions performed by the user. These examples are presented in order of increasing complexity.

2.4.1.3.1. Setting the Current Selection

This example illustrates a simple case in which the Word API directly reports the operation to be captured.

Most Word operations act upon the current S*election*. The current *Selection* can be a contiguous range within the document that the user has highlighted or can be collapsed into an *Insertion Point*. The user sets the current *Selection* by a variety of methods including: the directional arrows (left, right, up, and down) to move the insertion point, mouse clicks to set the *Insertion Point* at a determined position, dragging the mouse pointer to select a range, and expanding the current selection with Shift+arrow keys.

However, not all of these *Selection* changes need to be recorded. Those that are followed by another *Selection* change without an intervening document modification (e.g., when the user moves the *Insertion Point* several positions ahead by repeatedly pressing the arrow key) are superfluous and aren't recorded.

Capture Procedure

1. The Word API issues a *WindowSelectionChange* event and passes a reference to the Word *Selection* object each time that a different *Selection* is made.
2. The position of the current *Selection* is cached but not recorded until a subsequent document modification is detected.

2.4.1.3.2. *Command Bar* Bold *button (*B*)*

This example illustrates a case in which the MS Office API reports a GUI event that can be directly related to the operation to be recorded.

The Command Bar Bold button toggles the *Bold* property of the current *Selection*.

Capture Procedure

1. When a command bar button is pushed the Word API issues a *Click* event and passes a handle to the object that corresponds to that button. The event is triggered **before** the action associated with that button is executed.
2. One property of the Button object is its *Control ID*, a unique and fixed value assigned to each control. This ID is examined to detect the invocation of the *Toggle Boldface* command.

2.4.1.3.3. *Command Bar* Style *combo box (*Normal*)*

This example illustrates a case in which the document state resulting after a command has been completed must be examined to determine the parameters of an operation.

The Command Bar Style combo box sets the style of the current *Selection*.

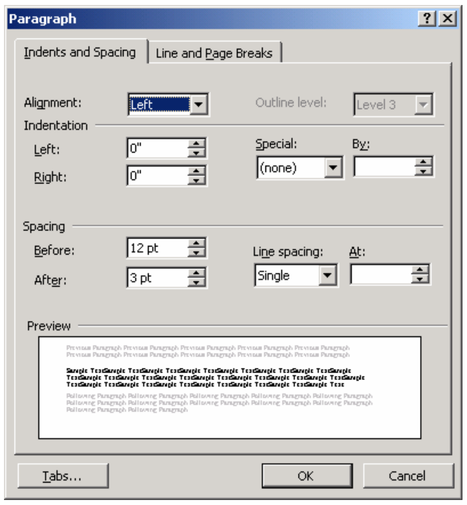
Capture Procedure

1. When the user changes the selection in a command bar combo box, the Word API issues a *Change* event and passes a handle to the Combo Box object as an argument. This event is triggered after the command associated with that control is executed.
2. The combo box's ID is used to identify the *Style* command. The style that the user selected is obtained from the *Style* property of the selected text (notice that this reflects the choice made by the user because the combo box Change event is triggered after the associated command is executed). A *ChangeStyle* operation is recorded with the name of the style as its parameter.

2.4.1.3.4. Format Paragraph *dialog box (Paragraph...)*

This example illustrates a case in which the document state before and after the execution of a command must be compared to determine the parameters of an operation. It also illustrates the use of a mediator to intercept the end of the processing of a command invoked through a menu.

The *Format* Paragraph menu item opens a dialog box for setting some paragraph properties of the current Selection (See Figure 1.).



Alignment: Left
Outline level: Level 3
Indentation
Left: 0"
Right: 0"
Special: (none)
By:
Spacing
Before: 12 pt
After: 3 pt
Line spacing: Single
At:

Figure 1: Format Paragraph Dialog Box

Capture Procedure

1. When the user selects the *Format Paragraph* menu item the Word API issues a *Click* event and passes a handle to the Menu Item object as an argument. This event is triggered before the action associated with that button is executed.
2. The menu item ID identifies the *Format Paragraph* command. The default values for the fields of the Format Paragraph dialog box are cached (this information reflects the state of the document before the execution of the command).
3. A mediator installed in the window message handler for the Menu Bar Pop-up window issues a pseudo-event signaling the completion of the procedure that processes this pop-up window. This event necessarily occurs after the *Format Paragraph* command has completed and is used as a surrogate for that event.
4. The pseudo-event triggers code that obtains the new default values for the fields of the *Format Paragraph* dialog box (which now reflect the state of the document after the execution of the command) and compares them with the cached values to determine which fields have changed. It records a *FormatParagraph* operation with those fields and their new values as parameters.

2.4.1.3.5. Copy/Move *through direct manipulation*

This example illustrates a case in which the Word API doesn't provide events that enable these actions to be captured. Hence we mediated the low level GUI interfaces to generate the required pseudo-events. To do so we took advantage of a visual clue in the GUI – that the mouse pointer changes its shape when it is over the selected text – to determine the nature of the application-level operation in progress.

In a direct manipulation Copy/*Move,* the selected text is dragged with the mouse to another location. If the *Ctrl* key is held down during this operation then it is a *Copy,* otherwise it is a *Move*.

Capture Procedure

1. A mediator installed in the window's message handler for the main Word window reports the beginning of the procedure that handles the *Mouse Left-Button Push-Down*.
2. The shape of the mouse pointer at this time is used to determine whether the mouse pointer was over selected text. If so and the left mouse button is depressed then a *drag* operation

has been initiated. The current location of the selected text is cached (this location corresponds to the state of the document before the *drag* operation).

3. The same mediator mentioned in step 1 reports the completion of the window message handler procedure, indicating the completion of the application-level *Copy*/*Move* operation.
4. Several elements of the current state of the document are examined to gather all the information needed to record the *Copy* or *Move* operation. First the state of the *Ctrl* key is checked to distinguish between *Copy* and *Move* operations. Then the now current location of the *Selection* is obtained (it now corresponds to the state of the document after the operation was carried out). Because Word has an *AutoCorrect* feature that automatically inserts or removes extra white spaces around the copied/moved text for the user and because this feature is only active for *Copy/Move* operations performed through the GUI (as opposed to those performed through the scripting API), the effects of this *AutoCorrect* operation must also be recorded for the current *Selection* and, in the case of the *Move,* in the place where the moved text used to be. Finally, the *Copy* or *Move* operation is recorded with parameters indicating the distance (positive or negative) that the selected text was moved and the number of surrounding white spaces added or deleted.

2.4.1.3.6. Typing (and AutoCorrect*)*

This example illustrates another case in which the Word API doesn't provide events that enable these actions to be captured. However, in this case there were a number of extra challenges. The first was that at the time that our mediators reported the typing activity, Word's internal state hadn't been updated so we couldn't use its API to capture the typed text. The second challenge was that capturing the typed text was only part of the job. By some obscure mechanism, Word divides long sequences of typing into undoable chunks (i.e., each *undo* operation undoes exactly one of these chunks of text). We therefore needed to detect exactly when Word chunked the typed sequence so that our recorded sequence of application-level changes would remain synchronized with Word's sequence of undoable units. We were able to overcome these problems through a discovery made after lengthy spying. We observed that Word redirected a keyboard message to a window message handler procedure only when it regarded that typed character as the first character of a new undoable unit. This gave us a way to detect the creation of those typed chunks. We also observed that when Word processed that first character of an undoable unit, its internal state had been updated to include the text typed in the previous chunk. This gave us a way to capture the typed text in each chunk.

Capture Procedure

1. A mediator installed in the window message handler for the main Word window reports the beginning of the procedure that handles the *Key-Down* messages. This event signals the beginning of a typing chunk.
2. The previous typing chunk, if any, is obtained through the Word API and recorded.

These two steps are repeated until a different type of event is received. This indicates that the last typing chunk is completed, and that last typing chunk is obtained through the Word API and recorded.

2.4.2. Replaying the Recorded Change History

This section describes the challenges faced in replaying the recorded change history to rebuild a corrupted document.

For most operations this replay is straightforward because the Word API has an operation that corresponds to the recorded change and the history includes all the information required for performing that operation.

However, for some operations (e.g. *AutoCorrect*) there is no corresponding operation in the Word API. For these operations the recorded change either had to be translated into a sequence of available operations, or the captured operation had to be simulated through the GUI so that Word would process it (performing the COM unavailable operation) as if it had come from the user. Sometimes this simulation through the GUI required execution of functions at the OS level to emulate device operations (e.g., keyboard input).

2.4.2.1. Replay Capabilities provided by the Word API

Word's COM interface provides the following functionality that facilitated execution of the captured operations (used for reconstructing a corrupted document).

- Methods to perform application level commands (e.g., Execute the *Toggle Bold Face* command on the selected text)
- Methods that perform fine grained changes to the elements that represent the state of the document (e.g., set the *boldface* property of the text between the positions 1 and 10 equal to True).

2.4.2.2. Document Modification Replay Examples

In this section we describe how the example document modification operations (described above in the Capture section) are replayed. The first three of these examples (Selection change, Command Bar Style combo box, and Format Paragraph dialog box) are skipped because the Word API contains an operation corresponding to these changes and replaying them merely consists of invoking those operations with the recorded parameters.

2.4.2.2.1. Copy/Move *through direct manipulation Replay*

This example illustrates a case in which the Word API does not provide a specific method for replicating this operation and hence it had to be implemented through a sequence of lower level commands. This forced us to create a special mechanism to handle the *undo* of these multiple-step operations atomically.

Although Word does not provide a specific method for *Copy* or *Move* they can easily be implemented through a sequence of lower level Word operations (copying, cutting, pasting, and inserting and deleting white spaces around the copied or moved text).

However, this multi-step implementation breaks the correspondence we've been maintaining between the recorded history and Word's undoable units. What was originally an atomic operation has, in the reconstructed copy, become a sequence of operations. If the recorded history contained an *Undo* of this operation, that *Undo* would operate differently in this reconstructed copy than it did in the original document (in the original document it would have undone the entire *Move* or *Copy* operation, while in the reconstructed copy it would only undo the very last operation in the multi-step sequence).

We handled this problem by placing special *multi-step-begin* and *multi-step-end* marks in the *Undo* Stack before and after executing this operation. When our implementation of the *Undo* operator pops out this special *multi-step-end* mark, it realizes that it is undoing a multiple step operation and keeps executing undoes until it reaches the *multi-step-begin* mark. To place these special marks in the *Undo* stack we inserted a special piece of text in the document ("<beginning multiple-steps operation>") and then deleted it. Hence, we only left a footprint in the *Undo* stack.

2.4.2.2.2. Typing (and AutoCorrect*) Replay*

This example illustrates a case in which the Word API does not provide an exact method for replaying this operation. Although it provides a mechanism for inserting text into a document, this action does not always produce the same effect as when the text was originally typed through the GUI because the API operation does not trigger the *AutoCorrect* and *AutoFormat* rules built into Word.

We overcame this problem by implementing our own typing method that simulated the entry of this text through the GUI to Word as if a user was typing it. However, this technique had two undesirable side effects that required compensation. The first was a synchronization issue. When replaying the history of operations to reconstruct a corrupted document, a race condition was produced between the operations executed through the Word API and the typing operations fed through the simulated keyboard input. This race condition was remedied through synchronization primitives that sequentialized the two input streams.

The second side effect from replaying our typing operators was Word would split the input into different undoable chunks than it had originally. To compensate for this problem we employed a grouping mechanism similar to the one described in the implementation of the *Copy/Move* operation above

2.4.3. Ensuring Document Integrity

When a document is created the Integrity Manager creates a version history for it. Each time that a persistent copy of the document is saved, the Integrity Manager generates a universally unique ID to identify that version of the document and stores this ID within the document itself. It also adds a record to the document's version history that contains the unique version ID, the user that saved the document, a timestamp, the captured sequence of the changes performed to the document since the previous version, and a cryptographic integrity mark that is a function of the content of the document (i.e., a cryptographic checksum).

When a document is opened, the Integrity Manager finds its version record in the document's version history and checks the integrity of the document against the cryptographic integrity mark stored in the version history. If the document was corrupted or was modified outside of the control of the Integrity Manager then it will no longer match its cryptographic integrity mark. When this occurs the user is offered the option of reconstructing the corrupted document from its recorded change history.

2.4.4. Attribution

The availability of the history of changes performed to a document allowed us to implement a novel attribution scheme. It utilizes a *time lever* to allow the user to move forward or backward through the document's history while looking at its state at those points in time. At each instant, the system displays the attribution information associated with the current operation (i.e. who performed it and when the operation occurred). In this time lever tool, forward transitions are performed by executing the recorded sequence of changes while backward transitions are performed by executing Word's *Undo* command.

2.4.5. Managing the Complexity of Word

As noted above, capturing all of the changes to a document is the biggest challenge faced by this project. It is exacerbated by the size and complexity of the Word interface. Based on an analysis of Word's built-in tool for customizing its user interface there are over 1100 unique Word commands and almost 900 unique command bar controls. This includes a broad range of commands from those that change the text font to commands that move the insertion point, to print command, etc. Of these, 209 affect the textual content of a Word document (i.e., excluding commands for drawing, manipulating forms, databases, pictures, webs, and tables).

2.4.5.1. Generic Capture and Replay

Given the formidability of the Word API, we recognized from the start of the project that individual capture and replay mechanisms could not be built for every Word command. Therefore we concentrated in the discovery of generic mechanisms that would allow us to accelerate the coverage of the totality of our target set of Word commands.

We succeeded in finding three generic mechanisms that combined covered 78% of our target set of commands. The first generic mechanism is supported by a macro execution facility that Word provides. This facility allows the execution of a command by passing its name as an argument. Using this facility we were able to implement a generic procedure for handling parameterless commands that act upon the current selection. To capture the command executed by the user this procedure maps the text in the Undo GUI control that describes the last executed command (e.g. Decrease-Indent) to its corresponding macro name (e.g., DecreaseIndent). To replicate the execution of these commands this procedure applies these commands to the current selection using the macro execution facility mentioned before.

The second generic mechanism is used to handle changes in the formatting attributes of the selected text. It is based in that a few formatting dialog boxes are able to consolidate all the changes that can be performed by a significant number of different Word commands. For example, there are several commands for making different changes to different font attributes (e.g., italics, underline, color, size) but all these changes can also be performed through a single dialog box, the Font dialog box. This procedure also takes advantage of a Word facility that allows inspecting the values that Word would load into a dialog box, changing these values, and executing these changes, all of this without ever displaying the dialog box to the user. To capture the changes performed by the user this procedure compares the values of these special dialog boxes before and after the execution of the user changes. To replicate the execution of these changes this procedure loads the new values into the dialog boxes and executes them.

The third generic mechanism took advantage of the versatility of the built-in Copy and Paste facility to implement a single procedure for handling the insertion and modification of most of the object types supported by Word (e.g., symbols, captions, pictures, dates, cross-references). Instead of implementing object type specific procedures for determining the object being inserted and for replicating the same operation at document recovering time, we implemented a generic procedure that copies the inserted object into the clipboard, saves the content of the clipboard in its binary representation, and pastes the saved content of the clipboard during document recovering.

The above strategies also required finding a general mechanism for detecting that the document had been modified that was independent of the particular command that caused that change. Initially, we planned to use Word's *document-modified* attribute for detecting these generic changes, but this would have required us to continually reset it through the scripting API to *not-modified* so that we could detect when generic (i.e. otherwise undetected) changes had occurred and to restore it when Word actually needed to access this document attribute (i.e. for *Save* and *Auto-Save*).

We have since discovered an alternative generic change indicator that is better because it is more specific and because it only relies on monitoring the state of the document (rather than changing it as we would have had to do with the *document-modified* attribute). This alternative generic change indicator is the depth of Word's *Undo* stack. Each time the document is changed, Word pushes the change (in a proprietary inaccessible format) onto the *Undo* stack so that it can later be undone if the user so chooses. While we can't access the change itself from the *Undo* stack (a real pity – being able to do so would have greatly simplified the Capture challenge), we can detect the presence of the new item at the top of the stack, and that is sufficient to determine that the document has been modified.

3. Safe Email

3.1. Scope

We developed a wrapper that monitors the run-time behavior of opened email attachments to ensure that these processes don't do anything harmful. It does so by detecting violations of process-specific rules establishing the acceptable (and safe) behavior of these processes relative to four resources: the file system, the system registry, inter-host communication, and process spawning.

The wrapper can determine whether an operation is being performed by the native application or by active content within the email attachment and applies a different (and presumably more stringent) set of rules to the latter.

When attempted violations are detected, the user is notified, informed of the severity of the violation, and determines whether to allow or prohibit the offending operation. The violation, the user's response, and the initiating email and attachment – obtained from the email client – are logged.

3.2. Approach

By monitoring the actual behavior of a running product, wrappers can ensure that security policy isn't violated – independent of how the product was written and whether or not it was corrupted during development by a design or programming error or hijacked at run-time by malicious code. If violations are attempted, they can be blocked or the application can be terminated.

This task utilized this wrapper technology to assure the safety of email attachments, even when those attachments contain active content such as macros, scripts, and executable object code.

Email attachments, like other imported content, pose significant risks to the state and persistent content of the machine on which they are opened. This risk is heightened by the fact that even attachments coming from trusted colleagues may have been unknowingly corrupted on their machine or even composed and transmitted without the knowledge of those trusted colleagues.

The source of this risk is the active content that can be embedded in documents delivered as email attachments[1]. While this active content can be beneficial (e.g. dynamically highlighting relevant material, automating repetitive steps, animating an image, or customizing an operation), it can also be quite harmful (e.g. overwriting or deleting files, altering the machine's startup configuration, or disseminating sensitive information).

This active content is a program written in some executable or interpreted language. Depending on the embedding document, it may be packaged as a macro, a script, or compiled code.

Three main techniques are currently used by COTS applications to control this active content. The first is a switch in an application that accepts active content that enables or

[1] It should be noted that while this description focuses on the danger arising from active content in email attachments, that same danger arises from active content (such as java or HTML script) in the email body itself. As explained in the Handling Active Content section, our approach handles this danger as well.

disables that active content. This binary switch either allows the active content (both good and bad) to execute or not. When the active content is disabled, the document can be safely opened.

The second technique is static analysis of the active content as used by virus-scanning software. This technique spots known malicious code sequences, marks the document containing them as infected, and prevents those documents from being opened until the document has been repaired (which the virus-scanning software can sometimes do itself). When this static analysis fails to detect the presence of any known malicious code sequences, the executables or documents with active content, can be opened with a higher probability of safely.

The third technique is a virtual machine, as exemplified by the Java Virtual Machine that is constructed to eliminate many of the sources of harmful behavior and to provide authorization mechanisms to control potentially harmful behavior. In this technique the designer of the virtual machine identifies the set of potentially harmful behaviors and enables a user provided authorizer function to determine whether or not to allow each invocation of these potentially harmful behaviors to occur.

We utilize a fourth technique – run-time monitoring and authorization – to ensure that active content executes safely and that any harmful behaviors are blocked.

Run-time monitoring and authorization is used to ensure that a process spawned to open an email attachment doesn’t perform any harmful behaviors. This monitoring and authorization is accomplished by mediating the interfaces used by these processes to access and modify resources. Harmful behavior is characterized as an attempt to access or modify an unauthorized portion of a resource as specified by the safety rules for that resource. Each invocation of a mediated interface is checked to see whether that invocation would violate these safety rules. If so, the invocation is blocked; otherwise it is allowed to proceed.

Thus, unless a process’s behavior violates one of the specified safety rules, that behavior is unaffected by the run-time monitoring and authorization. The safety rules act as a governor on the process’s behavior, prohibiting any actions that would harm – as defined by the safety rules – the protected resources.

3.2.1. Assumptions

This behavior monitoring approach rests on several assumptions:

1. All interfaces for accessing and modifying a resource can be identified
2. All of those interfaces can be mediated
3. The resource can’t be accessed or modified except through those mediated interfaces
4. Safety rules can be written for those resources that identify which portion of those resources can and can’t be accessed and modified.
5. Mediators can be written for those interfaces that determine whether or not the current interface invocation violates those safety rules

The difficulty of satisfying Assumption 1 is a function of the complexity of the service or platform providing the resource and the interfaces to that resource. For Windows NT and Windows 2000 (our target platforms), this can be quite challenging. For instance, NT and 2000 contain a half-dozen interfaces for opening or manipulating files as a unit (aside from those that change the contents of the file). Many of these interfaces also have an “extended” version (giving more control over the operation), and most have both an ASCII and a UNICODE version – thus quickly multiplying the set of interfaces that must be mediated.

Assumptions 2 and 3 are handled by the underlying wrapper infrastructure (see Underlying Wrapper Infrastructure section).

Assumption 4 is the critical assumption. It is discussed in detail in the Safety Rule Design section.

Good programming can satisfy assumption 5. The mediators must embody the semantics of the interface being mediated and express them in terms of the restrictions in the safety rules (e.g. that for Copy-File read access is required for the first (source) file and write access is required for the second (destination) file). The difficulty of the programming task is a function of the complexities of the interface and the safety rule language.

3.2.2. Handling Active Content

Because these protections are afforded by run-time behavior monitoring and the interfaces for accessing and modifying resources are defined by the operating system, it doesn’t matter how the processes are implemented – they can be written in any higher-level or assembly language. Access to the source code isn’t required, and the code can even be self-modifying – because no attempt is made to analyze the code itself to determine what it might do. Only the actual behavior of the process is monitored.

Thus, active content isn’t a problem – it is just additional code dynamically added to the native code base of the process. All this additional code can do relative to the protected resources is to make additional calls on the access and modification interfaces to those protected resources, or cause the native code to make additional or altered interface calls.

Since each of these interface calls will be monitored and authorized according to the specified safety rules, whether invoked directly or indirectly by the active content or by the native process code, the resources retain the same protection they had prior to the introduction of the active content.

What is different is that the intent of the active content cannot be presumed to be benign. It may be malicious, and if it is, it will attempt to perform harmful actions.

In our approach, the only brake on those harmful actions is the set of prohibitions contained in the safety rules that restrict which portions of the resources can be accessed and modified. Thus, it is essential that the safety rules accurately and completely identify these restrictions. If the safety rules don’t restrict an action, then malicious code can employ that action.

By applying these safety rules to the behavior of the email client process itself, these same protections are provided from any active content in that process – active content (such as HTML or Java Script) contained in the body of email messages and activated when those messages are read.

3.3. Accomplishments

The SafeEmailAttachments system protects the computer on which it is installed from malicious email attachments of all kinds including scripts and executables. It also protects the computer from malicious content in the body of email messages. This protection, in the form of a wrapper that encapsulates the execution of a process starts on the email client and is propagated to each process spawned by that wrapped process to open, or execute, an attachment. It is similarly propagated to any processes spawned by those processes, thus protecting the computer from all descendent processes spawned to open/execute an email attachment that might be malicious.

These protections have also been applied to other processes that might process malicious content or executables such as web browsers, document editors/viewers, media players, command processors, archive utilities, or the Windows shell that includes the File Manager and the Windows GUI.

Our SafeEmailAttachments system was extended with a *Contained Execution* facility that allows the execution of operations that violate security rules to safely continue by virtualizing their operation so that the effects are contained within the executing process and are not visible to any other process.

Its use ensures that processes run within their prescribed security policy (relative to the real resources protected by that security policy) without user intervention or control. Violations of that security policy are automatically and transparently contained without user involvement or knowledge.

This Contained Execution capability can be used to create a desktop user environment that is inherently safe for executing imported content of all kinds by limiting and managing the propagation of effects from these potentially malicious processes to the rest of the system. Several important classes of applications (such as viewers, importers, and editors) can be run under a single safety policy that contains all effects except for explicitly authorized information extraction (via the built-in copy-and-paste or SaveAs operations). This fully autonomic safety policy eliminates the configuration problem that has plagued existing behavior monitoring systems

The SafeEmailAttachments system was coupled by DARPA with Secure Computing's Autonomous Distributed Firewall to form a *Hardened Client* that – under separate DARPA funding – participated in the Navy's Fleet Battle Juliet exercise and in a test deployment to a Marine platoon stationed in Korea.

It was subjected to three separate Red-Team exercises with only a single unknown vulnerability discovered by those three Red-Teams.

3.3.1. Limitations

The underlying wrapper infrastructure is implemented for the Windows NT and 2000 platforms. Therefore this SafeEmailAttachment wrapper is only available on those platforms.

The SafeEmailAttachment wrapper handles both Outlook and Eudora as email clients. However, because the Eudora COM interface doesn't support the determination of the current message and attachment, that context cannot be passed to the spawned process's wrapper. Though we haven't applied it to other email clients (such as

Netscape Communicator and Outlook Express), it should work with them modulo the need to custom code the COM interface (if any) to determine the current message and attachment.

For some applications, Windows will use an already-running copy in preference to a fresh copy, and in those cases the already-running copy will retain its current wrapped or unwrapped status (i.e. if it had been previously started by opening an email attachment then it will already be wrapped and the system would be protected from the active content in the current attachment. However, if it had been started by the user via the menu or by opening a local document, then it would be unwrapped and the system would not be protected from the active content in the current attachment).

3.3.2. Comparison with Existing Practice

By far the most prevalent technique for protecting PCs from viruses, worms, and Trojan Horses is Virus Scanning software. As the name implies, this software scans (i.e. statically analyzes) documents and executables for known malicious code sequences (called signatures). Currently, the major vendors of these virus-scanning products have databases containing over 40,000 signatures and that number is increasing rapidly as additional malicious code sequences are found in the wild.

One advantage of our approach is that by monitoring run-time behavior and expressing the safety rules in terms of the portions of protected resources that each process can access and modify, its protection should be robust to new malicious code (including self-modifying viruses). This is in marked contrast to virus scanning software where maintaining protection in the face of new malicious code is the major shortcoming.

3.3.3. Wrapper Realization

We implemented SafeEmailAttachments as a wrapper in our NT Wrappers system [Balzer and Goldman]. The wrapper contains all the code for the mediators and the list of interfaces that these functions will mediate. This wrapper is placed around the email client program and is then propagated to each process spawned by that email client to open an email attachment.

The wrapper protects four types of resources: the file system, the system registry, the set of hosts that can be communicated with, and the set of processes that can be spawned. The safety rules identify which portions of these resources can be accessed and modified.

3.3.3.1. Underlying Wrapper Infrastructure

Much of the functionality of the SafeEmailAttachments wrapper, and all other wrappers, is supplied by the underlying wrapper infrastructure.

First, it installs the SafeEmailAttachments wrapper around the Email client whenever it is started. This occurs through the shortcut used to start the Email client.[2] The shortcut runs a wrapper infrastructure program – SpawnWrappedProcess – that spawns a named process (here the Email client) encapsulated with a specified wrapper (here the SafeEmailAttachments wrapper).

Second, it propagates this wrapper to every process spawned by the Email client. These processes are spawned to handle the opening of an email attachment (either the application registered to handle that type of email attachment, or if the attachment is an executable, then the attachment itself as a process). These are precisely the processes that need to be wrapped to ensure the safety of opening those email attachments.

The wrapper infrastructure provides this service because most wrappers need to also wrap any processes that are spawned by the process they are wrapping so that the same protections are provided for the execution of those child processes (i.e. a malicious program can't escape a wrapper's encapsulation by spawning a child process to do the dirty work). By incorporating this service in the wrapper infrastructure, wrapper development was simplified and a potential source of errors, whether by omission or faulty coding, was eliminated.

Finally, and most importantly, the wrapper infrastructure ensures that the wrapped process can neither remove the wrapper nor bypass the mediators it contains. This "NonByPassability" is crucial for all wrappers – without it the interfaces being mediated could be invoked unobserved and unregulated, and the wrapper would be unable to guarantee any of the protections for which it was designed.

3.3.3.2. Safety Rule Design

For each type of protected resource (currently the file system, the system registry, communication with other hosts, and spawned processes), the safety rules identify which portions of those resources can be accessed and which portions can be modified.

Each rule is a path expression with wild cards (the asterisk character matches any sequence of characters). Each Access and Modify specification consists of an Allowed and a Prohibited group of rules.

For a mediated interface invocation to be authorized each resource use it entails (Access or Modify) must satisfy at least one of the rules in the corresponding Allowed group and none of the rules in the corresponding Prohibited group.

Thus with a file system rule specification of:

```
;; AllowedFileReadAccessRules
*
;; ProhibitedFileReadAccessRules
*/Microsoft/Address Book/*
;; AllowedFileModifyRules
```

[2] In our "lightweight" distribution of the SafeEmailAttachments wrapper, we rely on the user starting the email client wrapped – via the shortcut provided. If they start the email client without this wrapper, whether by accident or on purpose, none of the protections will be provided. To protect against such occurrences we have a Policy Manager that ensures that the appropriate wrappers are placed on every process that is started, but this "heavier-weight" component is not packaged with our SafeEmailAttachments distribution.

```
*/Microsoft/Office/*
*/Microsoft/Word/*
*/temp/*
;; ProhibitedFileModifyRules
*.dot
```

would allow the mediated process to read files anywhere in the file system (the "*" rule) except for those files in any "Microsoft/Address Book/" subdirectory. It could modify files only in "Microsoft/Office/", "Microsoft/Word/", or "temp" subdirectories, but not those with a ".dot" extension.[3]

3.3.3.3. Process-Specific Safety Rules

The (simplified) example rules above were obviously tuned for a particular process – Microsoft Word – that needs to be able to modify files in particular directories that it manages and to operate on particular types of files in those directories.

Most COTS programs have similar requirements for portions of a protected resource that they need to be able to access and modify to function properly. We have therefore established separate safety rule sets for each application so that the safety rules can be designed to allow only the access and modification rights to the protected resources required by that application.

With such "focused" application-specific safety rule sets, malicious code within a mediated process becomes much easier to spot because it deviates from the focused protected resource subsets used by the original native mediated process.

These application-specific safety rules are not wired into the SafeEmailAttachments wrapper. Rather they are loaded as the wrapper is initialized. The wrapper is thus both data driven (because the safety rules are dynamically loaded) and polymorphic (because the set of safety rules is selected on the basis of the process being wrapped).

3.3.3.4. Active Content Safety Rules

The application-specific safety rules described above are designed to allow a process to access and modify all, and only, the protected resources it needs for its proper operation. Thus, they are not designed to allow access to any other protected resources that might be required by active content embedded within an email attachment opened by this process. Nor are they designed to protect the resources used by this application from that active content.

Separate safety rules have therefore been established for any active content that is embedded in email attachments opened by a wrapped process. The SafeEmailAttachments wrapper can distinguish, by examining the call-stack (to determine whether control is dynamically within the Interpreter module for the active content), between resource interface invocations made by active content and those

[3] These file safety rules are also used to protect all the files used by the wrapper such as those containing the safety rules. Prohibitions against reading or writing these files are silently added to the rules as they are loaded (to prevent these prohibitions from inadvertently being omitted from the safety specification).

made by the native process, and it uses this distinction to dynamically switch back and forth between the original safety rules for the process and these separate safety rules for the embedded active content within any email attachments opened by that process.

3.3.3.5. Advanced Safety Rules

Two refinements to the safety rule language simplify the specification of the safety policy for an application. The first is the addition of an annotation to a file modify rule that limits its application only to files created by that process. Applying this annotation prevents the application from modifying any preexisting files that would otherwise match the rule specification, but allows it to create and then modify new files that would satisfy that specification.

The second is a default response to a read-write file open operation when the safety rules allow the file to be read but not modified. The default response is to automatically deny the read-write file open without interacting with the user in the hope that the application will then attempt to reopen the file with just read privileges. Many applications behave exactly this way and the default response causes them to operate on the file as prescribed by the safety rules (i.e. with read-only privileges).

However, some applications will react badly to the denial of full read-write privileges. An override of this default is therefore provided so that the user can grant these additional privileges to the wrapped process.

Thus, by granting read, but not modify, privileges to the files in the email attachments directory, attachments are automatically opened in the processes wrapped by SafeEmailAttachments with read-only privileges while they would otherwise be opened with read-write privileges. Similarly, this default can prevent templates from being updated every time a document is opened.

3.3.4. Handling Safety Rule Violations

When safety rule violations are detected an alert is displayed for the user identifying the offending process, the type of resource involved (file, registry, remote host, or process being spawned), the particular resource involved, and the prohibited type of access attempted (read, write, download, upload, or spawn).

The user can choose to prohibit the attempted operation, allow it to occur (overriding the safety rule), or abort the offending process. If the user chooses to prohibit the attempted violation, an access-denied error (as defined by the API of the particular offending resource interface) is returned to the caller. Most COTS programs are prepared to receive such errors (as resources may be exclusively in use by another process, offline, or otherwise unavailable), but some aren’t and this error may cause them to terminate.

To eliminate the need for further user interaction on any repetitions of a safety rule violation, the user’s choice is cached for the duration of the process and automatically reapplied if the process repeats the same safety rule violation.

3.3.4.1. Need For Context

As part of an experiment[4] to integrate this email attachment safety into a broader cyber-defense system [Musman and Flesher] that orchestrated email attachment violations into revised distribution policies for the email server, we had to create a log indicating the nature of the safety violation and the email and attachment that initiated that violation.

The first portion of this was easy because the violation was readily available and had already been converted into a comprehensible form (for display to the user in the safety alert). However, much to our surprise and chagrin, we discovered that the initiating email and attachment were not available. That information was in a separate process – the email client that had spawned this process. Furthermore, this information could no longer be obtained from that process – it was only available in the email client at the point at which the attachment-handling process was spawned.

We therefore revised our SafeEmailAttachment wrapper to obtain this information from the email client (via its COM interface) when an email attachment is opened in anticipation of its possible need by the attachment-handling process if a safety violation subsequently occurs in that process. This “context” information is then passed in the wrapper parameter[5] to the spawned process’s SafeEmailAttachment wrapper so that it is available in that process if needed.

3.3.4.2. Alert Severity

A small database of rules identifying especially dangerous safety violations and describing the nature of the danger was added to the SafeEmailAttachments wrapper to alert the user to these situations. Determining whether the safety violation was dangerous required additional context beyond just the resource being accessed and the type of access attempted required for detecting safety violations. This context was expanded to include the operation being performed, all of its parameters, and the attachment being opened.

One example of how this additional context is used to determine whether a safety violation is dangerous is the rule that if the safety violation is an attempt to write a file in a protected area and that file is a copy of the attachment itself then the violation is dangerous because the attachment is trying to replicate itself.

A second example of how this additional context is used to determine whether a safety violation is dangerous is the rule that adding the attachment to the set of programs to be run at startup is dangerous because the attachment is trying to ensure it is run whenever the machine is booted.

4 With Scott Musman (IMSC) in DARPA’s Autonomic Information Assurance program

5 The existing wrapper parameter mechanism was generalized to allow the value of the wrapper parameter for a child process to be dynamically determined by the spawning parent process

3.3.5. Contained Execution

3.3.5.1. Wrapper Configuration Problem

Reference Monitors – mediators (such as those used in SafeEmailAttachments) that monitor all references from a program to the resources it uses to ensure they satisfy the access control policy being imposed on the program (sometimes referred to as *Wrappers* or *Sandboxed Execution*) – provide protection by blocking malicious behavior before it occurs. They work for both known and novel attacks because they are blocking malicious behavior rather than examining the code. But Reference Monitors face two critical problems

1. They must be carefully configured to identify all the malicious behavior to be blocked,
2. They are subject to generating false positives as non-malicious programs violate the hard boundaries established in defining malicious behavior.

Currently all Reference Monitors operate strictly as authorizers, either allowing or prohibiting particular operations on the basis of whether or not they violate the security rules being enforced. This authorization function operates exactly like the operating system's authorizer, except that the developer of the Reference Monitor rather than the operating system vendor chooses the set of resources that can be secured and the set of policies that can be applied to those resources.

The problem with this authorization approach is that a non-retractable binary decision must be made (at the point that the operation is being performed) before enough information can be gathered for making that decision. In the absence of sufficient information, safety dictates conservative decisions that prohibit the potentially malicious operation, thus generating many false positives for non-malicious programs. These false positives, in turn, cause the user to either ignore the security system or turn it off.

3.3.5.2. Contained Execution

A new technology developed under this contract, called *Contained Execution*, eliminates these two problems with Reference Monitors by introducing a new option for security rule violations. This new option allows the execution of operations that violate security rules to safely continue by virtualizing their operation so that the effects are contained within the executing process and are not visible to any other process. The transparency of this virtualization allows the process to continue its execution unaware that some of its actions have been contained within the process. As far as it can determine, those operations have been successfully performed.

As shown in Figure 2, this virtualization is accomplished by redirecting the offending operation from its original resource *X* to a newly created resource *Y* that is only accessible to the contained process. The operation is performed on this newly created resource *Y* and subsequent operations by the contained process on resource *X*, whether they violate the security rules or not, are redirected to resource *Y*. So, for example, if the contained process attempts to modify a file *X* that would violate the security rules, then a new file *Y* is created in a reserved portion of the file system and initialized to the contents of the original file *X* whose contents can't be modified. The offending modification is then performed on the newly created file *Y* as are all subsequent operations on the original file *X*. Thus, the contained process operates on the substituted file *Y* and can read the modifications it is making to that file, but all other processes that access that file operate on the original (and unmodified) file *X*.

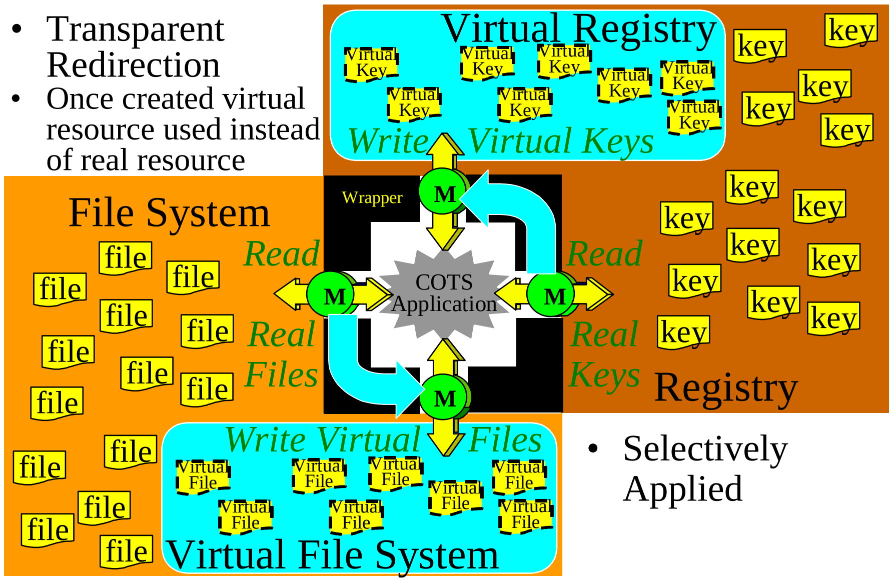


Figure 2: Contained Execution Through Transparent Redirection

Virtualization of other resources such as the registry occurs in exactly the same way. When the contained process terminates, these virtualized resources are destroyed. Thus, all of the contained process' effects that violate the security rules are contained within its execution and don't affect the execution of any other processes running concurrently or subsequently.

Contained Execution is like virtual machines [Steel] in ensuring that changes are made to virtual resources not visible outside the virtual machine. However, unlike virtual machines it operates at the level of an individual process rather than a whole machine including its operating system. In addition, rather than having to copy the entire set of resources available to the virtual machine prior to execution, Contained Execution is incremental, only copying resources as they are contained.

3.3.5.3. Benefits of Contained Execution

One immediate benefit of Contained Execution is that when a wrapped process violates a security rule, the user (or automated agent) is not forced to make a non-retractable choice between unsafely allowing the offending operation to proceed (thus overriding the security violation), conservatively prohibiting the operation (but altering the process's execution by preventing the operation), or, most conservatively, terminating the process.

Often, the user (or automated agent) doesn't have enough information to make this choice at the point at which the security violation occurs. There may not be an obvious mapping between the particular security violation and the function being performed by the process. In addition, the process may have just started and the user can't yet determine whether the process is functioning as expected or performing anomalously. Moreover, users find responding to such individual interactive alerts intrusive and objectionable.

With Contained Execution, the security violation can be *contained* allowing execution to safely proceed without any apparent alteration (aside from the containment of the effects of the operation to the contained process). This additional choice, because it is both safe and doesn't have any apparent effect on the process execution, enables the user (or automated agent) to delay deciding whether or not the process is malicious and its actions are harmful. In fact, this determination can be delayed until the process has terminated and all its behavior has been observed.

Once the process has terminated, the process's contained resources can be examined to determine whether or not the process was malicious. If it was, then the virtualized effects of the contained operations should be discarded so that they have no persistent effect outside that process. If it was determined that the process was not malicious, then the contained effects (or a subset of them) *could* be mapped back onto the system's real resources so that these effects became persistent and visible throughout the system.[6]

Moreover, since containment is safe and transparent to the process execution, containment can be automated on the user's behalf. There is no need to notify the user that security violations are being automatically contained.

Finally, to the extent that the desired effects of a process execution occurred during that execution without violating security policy (such as updating files it was authorized to modify or displaying information for the user), then the contained effects – which were contained because they did violate security policy – can be discarded without examination or the need to determine whether or not the process was malicious, because those effects were not the reason why the process was run, but merely unintended "side effects" of that process execution (such as saving backup copies, creating temporary files, etc).

This frees the user from having to examine a process's security violations, its contained effects, and determining whether it was malicious or not. Contained Execution ensures that processes run within the prescribed security policy (relative to the real resources protected by that security policy) without user intervention or control. Violations of that security policy are automatically and transparently contained without user involvement or knowledge.

The next section considers how aggressive use of Contained Execution can be used to ensure such autonomic and invisible safety for a variety of important classes of programs.

3.3.5.4. Autonomic Desktop Safety through Contained Execution

This Contained Execution capability can be used to create a desktop user environment that is inherently safe for executing imported content of all kinds by limiting and managing the propagation of effects from these potentially malicious processes to the rest of the system. Moreover, as described in the previous section, the security mechanisms that produce this safe desktop environment are fully autonomic and invisible to the user.

[6] While this option could be implemented, it has not been implemented and is unlikely to be implemented because of the current and expected use of Contained Execution explained in the rest of this paper.

This safe desktop environment is based on the recognition that the purpose of the vast majority of imported content is to provide information for the user. This is accomplished by presenting the imported content to the user through the computer's user interface or output devices, i.e. displaying that information graphically or textually, playing an audio track, and/or printing a copy for offline consumption.

3.3.5.4.1. Desktop Safety for Content Presentation Programs

However, with Contained Execution, these untrusted presentation programs can be safely used to produce the presentations needed for user access to the imported content. Contained Execution can be used to contain all other effects so that whether the imported content was malicious or not the only effect of executing the presentation program on the imported content would be the presentations of that content provided to the user.

Since these "presentation" programs are untrusted, it doesn't matter whether they come from a fixed predefined set, or are dynamically supplied (with the imported content or dynamically installed for it). Thus, any arbitrary program can be safely run in this environment. No matter what it does, the only effect it will have on the system is the information (i.e. presentations) it provides to the user through the system's output devices.

An important benefit of this use of Contained Execution is that it eliminates the configuration problem that has plagued all behavior authorization systems (such as wrappers, reference monitors, and even operating system access controls when applied to programs). Rather than carefully configuring a security policy for each application specifying which resources it can and cannot modify (and inevitably finding that it is incomplete or inaccurate because some infrequent but required behaviors are omitted), a single security policy is used for this entire class of applications. Moreover, specifying that security policy is extremely simple – *all modifications to any persistent resource are contained*.

It should also be noted that even when used in this all-encompassing mode, Contained Execution maintains its performance advantage over virtual machines because of its incremental copy-on-modify implementation – only those resources that are actually modified during execution need to be virtualized and copied.

User Extraction of Content from Contained Execution

Having read, seen, or heard something of interest in the imported content, users need ways of incorporating and integrating it with other information on their computer. To keep the security mechanisms hidden and transparent, users are able to use the standard operating system mechanisms for extracting information from a process (i.e. copy-and-paste and the SaveAs dialog). These are treated as user authorized exceptions to the process's containment policy and result in the modification of the real resource (clipboard or file) so that the extracted information persists and remains available after the contained process terminates.

To ensure the safety of these extraction operations, the extracted content should be stripped of any active content before being made available to any other process (via paste) or saved in the file system (via SaveAs).[7]

[7] This safety mechanism for extracted content has not yet been implemented.

3.3.5.4.2. Desktop Safety for Programs that Import Content

Some programs that import content also consume that content and present it to the user. Web Browsers are a prime example of this class. These combined import-and-presentation program can be safely executed with the precisely the same configuration as other presentation programs – namely that all their effects are contained except for data explicitly saved by the user.

But programs that merely import content without themselves presenting that content to the user must be handled a little differently. They need to persistently store that imported content so that other programs can present that imported content to the user. The user's authorization to extract this information from the contained process is implicitly contained in their authorization to import that content. For these content import programs the contain everything policy must be augmented with one or more areas (such as the browser cache or a "download directory") that the process is authorized to persistently modify (and thus perform information extraction to these areas on the user's behalf). This information extraction authorization can be limited to just adding information to these designated areas (thus preventing overwriting of any existing information contained in those areas) or also allow existing information in those areas to be updated (by a more recent version of some periodically imported page, file, or document).

3.3.5.4.3. Desktop Safety for Editors

Another class of applications that can be made inherently safe through Contained Execution is editors. They are very similar to presentation programs in that they enable the user to view or hear the content of a page, file, or document. However, their purpose is to enable the user to also modify that page, file, or document and save the new versions that are produced.

They can thus be safely executed with the same containment policy as presentation programs – contain everything except for explicit user information extraction (via copy-and-paste and SaveAs) – with the additional exception that the user is implicitly authorizing the editor program to persistently save new versions of the page, file, or document being edited.

3.3.5.5. Comparison with Existing Practice

Contained execution indeed isolates execution; in a sense it is providing a virtual machine – as originally conceived by Steel [Steel] and supported by commercial products such as *VMware* or *Virtual PC* – for the execution of the contained process. However, recent virtual machine efforts emphasize providing a new interface, usually more abstract and more convenient for solving some problem – such as efficient distributed computation [Geist] or statically checkable type safety [Lindholm] –, rather than just protecting the implementation platform upon which it runs. Naturally, because they isolate the underlying machine they still provide that protection, and there are even efforts to control the policies under which the platform is modified, e.g. security policies in the Java Virtual Machine [Gong]. Other goals of virtual machines include isolation of users from one another [Wallach] and provision of resource management facilities [Back].

The point of Contained Execution is instead, to appear to provide exactly the *same* machine as the implementation on which it is running. Emulation, sandboxing, and the original form of virtual machines [Steel], are well-known mechanisms for providing protection [Chang] through isolation of the execution resources from the platform's real resources. All of these mechanisms, hereafter collectively referred to as *sandboxing*, afford greater functionality than the simpler fine-grain access control (denial) mechanisms [Badger, Balzer, Fraser, Goldberg] found in Reference Monitors [Anderson] because they allow (virtualized) execution to continue rather than disrupting execution by blocking access to the protected resource. But *sandboxing* mechanisms are generally controlled on a per-user based policy [Dan] rather than affording the process-level protection that our contained execution provides. Moreover, we are not aware of any *sandboxing* mechanisms that incrementally virtualize resources as needed during execution.

An effort that is similar in spirit to our work is the "consh" system at UC Santa Barbara. This Unix-based system uses Janus mechanisms and confined execution principles to make Internet access transparent (but isolated) in the same sense as our mechanism isolates registry and file system access transparently [Alexandrov]. Because it is substituting for the Internet, it is necessarily incremental and also shares the property with ours that no kernel-level code is necessary to afford the protection. (It is not clear whether work is proceeding on this system.)

Of course, all of these efforts can be related to a general framework for deception [Cohen]. By pretending to be compromised – as contained execution does –, more can be learned about the activities of the compromiser. Although our mechanism now reflects whatever program changes were demanded, introducing spoofing mechanisms to modify "observables" would be an easy extension to the mechanism.

3.3.5.6. Limitations

3.3.5.6.1. Conceptual

Contained Execution only ensures the *safety* of program execution – that the program does not modify any persistent resources it is not authorized to change. Like other sandboxing techniques, this guarantee only applies to the particular program execution being controlled. Other executions of the program may attempt to modify additional or different unauthorized persistent resources.

Moreover, those attempted unauthorized modifications may arise either because the safety policy was misconfigured for the program and didn't authorize modification of all the required resources or because the program really was malicious. Distinguishing between these two cases can be difficult when the unauthorized resource modification is not obviously related to the program's operation.

This safety guarantee says nothing about either the correctness or non-maliciousness of the program execution – only that no unauthorized persistent resources were modified. Through a design or implementation flaw, a program may make incorrect modifications to authorized resources, or present incorrect information to the user. These same "flaws" could be introduced by malicious intent to corrupt persistent resources and/or mislead the user without violating the safety policy being enforced.

3.3.5.6.2. Implementation

The underlying wrapper infrastructure is implemented for the Windows NT, 2000, and XP platforms. Therefore this Contained Execution capability, which is implemented as a wrapper, is only available on those platforms.

Contained Execution has currently only been implemented for the file system and registry. Until Contained Execution has been implemented for additional classes of resources, such as devices and communication, our system will continue to protect those resources through its denial (prevention) mechanism.

4. References

[Alexandrov] A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations. Unpublished, 1998. (See http://www.cs.ucsb.edu/~berto/papers/index.html)

[Anderson] Anderson, J. P., Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA, 1972.

[Back] G. Back, W. Hsieh, J. Lepreau Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*.

[Badger] Badger, L., Daniel F. Sterne, David L. Sherman, and Kenneth M. Walker. A Domain and Type Enforcement UNIX Prototype. Vol. 9, *UNIX Computing Systems. Glenwood, MD*: Trusted Information Systems Inc. 1996.

[Balzer] R. Balzer and N. Goldman. Mediating Connectors. *In Proceedings of the 19th IEEE Conference on Distributed Computing Systems, Austin, TX*, May 1999, 73-77.

[Balzer and Goldman] Robert Balzer and Neil Goldman: Mediating Connectors: A Non-ByPassable Process Wrapping Technology, DARPA DISCEX Conference 2000, Hilton Head, SC. Jan 25-27, Vol II, pp 361-368.

[Chang] F. Chang, A. Itzkovitz and V. Karamcheti. Secure, User-level Resource-constrained Sandboxing. TR1999-795 Department of Computer Science, Courant Institute of Mathematical Sciences, NYU December 16, 1999

[Cohen] F. Cohen, D. Lambert, C. Preston, N. Berry, C. Stewart, and E. Thomas. A Framework for Deception . Fred Cohen & Associates. http://all.net.

[Dan] A. Dan, A. Mohindra, R.Ramaswami and D. Sitaram. ChakraVyuha (CV): A Sandbox Operating System Environment for Controlled Execution of Alien Code IBM RC20742. 1997.

[Fraser] T. Fraser, L. Badger, M. Feldman. Hardening COTS Software with Generic Software Wrappers. *IEEE Symposium on Security and Privacy.* 1999.

[Geist] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Mancheck, V. Sunderam, “PVM: Parallel Virtual Machine- A User’s Guide and Tutorial for Network Parallel Computing”, *MIT Press, 1994.*

[Goldberg] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications – Confining the Wily Hacker. In *Proceedings of the 1996 USENIX Security Symposium*, 1996.

[Gong] Gong, Li, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2, pp. 103-112 in *Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California.* Berkeley, CA: USENIX Association. 1997.

[Lindholm] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Addison-Wesley, 1997

[Musman and Flesher] S. Musman and P. Flesher: System or Security Managers Adaptive Response Tool DISCEX 2000 proceedings. Hilton Head, SC. Jan 25-27, Vol II, pp 56-68

[Steel] T. B. Steel. A first version of UNCOL. In Western Joint Computer Conference, May 1961

[Wallach] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In IEEE Symposium on Security and Privacy, May 1998.

5. Published Papers

The following papers that were published during this contract document the work performed on the various tasks and describe the accomplishments achieved:

[1] Marcelo Tallis and Robert Balzer, "Document Integrity through Mediated Interfaces", In the Proceedings of the Second DARPA Information Survivability Conference and Exposition (DISCEX II), June, 2001.

[2] Robert Balzer, "Assuring the Safety of Opening Email Attachments", In the Proceedings of the Second DARPA Information Survivability Conference and Exposition (DISCEX II), June, 2001.